Notes on "Algebra of Programming"

Glenn Strong
Department of Computer Science
Trinity College, Dublin
Dublin
Ireland
Glenn.Strong@cs.tcd.ie

December 15, 1999

1 Background

In the book "Algebra of Programming"[3] it is asserted that:

$$\langle sum, length \rangle \;=\; (\!|zeros, pluss|\!)$$

In the notation of Bird and de Moor, to be discussed later, each side of this equation describes the computationally expensive portion of a function to calculate the average (mean) of a list of natural numbers, under suitable definitions of *sum*, *length*, etc. The equation denotes the equivalence of two forms of this function, an efficient version and an inefficient version.

Each side indicates a function to produce a pair of numbers, such that when the first is divided by the second will be the average of the list. In functional programming terms there are three functions whose types are:

```
sum :: [Int] -> Int
length :: [Int] -> Int
sumlen :: [Int] -> (Int,Int)
```

The `sumlen` function corresponds to the $(\!|zeros, pluss|\!)$ form, and represents a function which will only traverse its argument once. Assuming the existence of some function `div` which divides pairs

```
div (a,b) = a/b
```

then the average of a list is

```
average = div (sum,length)
```

or, using `sumlen`,

```
average = div sumlen
```

This second form is a more efficient implementation provided that `sumlen` is expressed in such a way that it only traverses its argument once. Bird and de Moor claim that $(\!|zeros, pluss|\!)$ (which is more fully explained in section 2) is such a form.

An attempt to prove this requires one to understand and use much of the material from the preceding two chapters of the book. It provides, however, a valuable insight into the techniques for program derivation offered by Bird/de Moor. In addition, deriving an implementation of this provides a useful link to "Introduction to Functional Programming using Haskell"[1] in which the problem is also mentioned.

1.1 Notation

Chapter two of Bird/de Moor introduces the concepts and notation, based on Category Theory[4], which will be required to understand the proof. This section provides a brief introduction to the notation, and provides some of the laws. For details, including proofs of properties, consult chapter 2 of Bird and de Moor. In particular, many definitions and functors introduced in the book are omitted here as they are not directly used in to the proof.

- The terminal object of a category is written 1, and has the property that for all objects in the category there is a unique arrow from each object of the category to it.

- A pair of arrows can be written $\langle A, B\rangle$. Pairs satisfy the following "fusion law" for composition:
$$\langle f, g\rangle \cdot m = \langle f \cdot m, g \cdot m\rangle \tag{1}$$
The "cancellation laws" for these objects are:
$$\begin{aligned} outl \cdot \langle f, g\rangle &= f & (2)\\ outr \cdot \langle f, g\rangle &= g & (3)\end{aligned}$$
These introduce a pair of functors, *outl* and *outr* which are used to manipulate pairs.

- A product of two objects (A and B) is written $A \times B$. If each pair of objects in a category has a product, then $\times$ can be made into a bifunctor:
$$f \times g = \langle f \cdot outl, g \cdot outr\rangle \tag{4}$$

- Coproducts are written $A + B$, and the unique arrow from $A + B$ to C is written $[f, g]$. The pronunciation "case f or g" is suggested. The so-called "coproduct fusion law" is:
$$m \cdot [h, k] = [m \cdot h, m \cdot k] \tag{5}$$
The "exchange law" (the proof of which is left as an exercise in Bird/de Moor) is
$$\langle [f, g], [h, k]\rangle = [\langle f, h\rangle, \langle g, k\rangle] \tag{6}$$
This law can be imagined as representing a pair of selections; naturally, it will result in one of two pairs.[1]

- Type definitions are written in a style familiar to functional programmers. The natural numbers, for instance, can be written:
$$Nat \ ::= \ zero | succ\ Nat$$
A declaration of this form introduces an *initial algebra*, α; in this case it is $[zero, succ]$, of a functor F. The functor F is defined here as:
$$\begin{aligned} \mathsf{F}A &= 1 + A\\ \mathsf{F}h &= id_1 + h\end{aligned}$$

[1] In the interests of completeness the following proof of the exchange law should be sufficient:
= { By $m \cdot [a, b] = [m \cdot a, m \cdot b]$, (*p. 42* in Bird de Moor) }
$outl \cdot [\langle f, h\rangle, \langle g, k\rangle] = [f, g]$
$outr \cdot [\langle f, h\rangle, \langle g, k\rangle] = [h, k]$
= { $\langle outl, outr\rangle = id$, by definition of id }
$\langle [f, g], [h, k]\rangle = id \cdot [\langle f, h\rangle, \langle g, k\rangle]$
= { $id \cdot m = m$ }
$\langle [f, g], [h, k]\rangle = [\langle f, h\rangle, \langle g, k\rangle]$

(in Bird/de Moor a functor F maps both arrows to arrows and objects to objects, hence the two equations).

- Catamorphisms are written:

$$(\!|c, f|\!)$$

This is a shorthand for $(\!|[c, f]|\!)$ which drops the inner brackets. Catamorphisms can be used to represent iterative computational schemes. If h is some catamorphism on Nat: $h = (\!|c, f|\!)$, then

$$\begin{aligned} h(0) &= c \\ h(n+1) &= f(n, h\ n) \end{aligned}$$

See section 3

2 The List Average Problem

2.1 Definitions

A cons list over some type A is:

$$listr\ A\ ::=\ nil \mid cons\ (A, listr\ A)$$

which has an initial algebra $[nil, cons]$ of the functor

$$\mathsf{F}(A, B) = 1 + (A \times B) \tag{7}$$

$$\mathsf{F}f = id_1 + (id \times f) \tag{8}$$

while the type of natural numbers is:

$$Nat\ ::=\ zero \mid Succ\ Nat$$

The function *sum* which returns the sum of a list of natural numbers is:

$$sum\ =\ (\!|zero, plus|\!)$$

The function *length* which returns the number of elements in a list is:

$$length\ =\ (\!|zero, succ \cdot outr|\!)$$

We can define a function *average* which returns the median of a list of natural numbers:

$$average\ =\ div \cdot \langle sum, length \rangle$$

For completeness, let us define *plus* and *div* (which will have to be total):

$$plus(a, b) = a + b$$

$$\begin{aligned} div(0, 0) &= 0 \\ div(a, b) &= a/b \end{aligned}$$

2.2 The Banana Split Law

A naive implementation of this previous definition of *average* will result in a program which traverses its argument twice. Let us introduce the "banana-split law" which enables us to express any pair of catamorphisms as a single catamorphism.

$$\langle (\![h]\!), (\![k]\!) \rangle = (\![\langle h \cdot \mathsf{F}\ outl, k \cdot \mathsf{F}\ outr \rangle]\!) \tag{9}$$

The proof of this equation is on page 56 of Bird/de Moor.

Bird and de Moor assert that this equation can be used to convert the previous definition of *average* thus:

$$\langle sum, length \rangle = (\![zeros, pluss]\!)$$

where

$$zeros = \langle zero, zero \rangle$$

and

$$pluss(a, (b, n)) = (a + b, n + 1)$$

2.3 Proof

Bird and de Moor leave it to the reader to prove the equality of the two implementations of *average*. We will spend the remainder of this section proving this equivalence. Throughout the proof page and section references in italics are references to Bird and de Moor[3]. Note that a somewhat more abbreviated proof is available in the supplementary material to Bird and de Moor[2]. This more detailed exposition of the proof will hopefully serve to fully illustrate the process involved in creating category theoretic proofs in the Bird and de Moor system.

$$\langle sum, length \rangle$$

= { Definition of *sum* and *length*}

$$\langle (\![zero, plus]\!), (\![zero, succ \cdot outr]\!) \rangle$$

= { By the banana split law (eq. 9) *(p. 56))*}

$$(\![[zero, plus] \cdot \mathsf{F}\ outl, [zero, succ \cdot outr] \cdot \mathsf{F}\ outr]\!)$$

= { By the definition of F (eq. 7, eq. 8) *(p.49)* }

$$(\![[zero, plus] \cdot (1 + (id \times outl)), [zero, succ \cdot outr] \cdot (1 + (id \times outr))]\!)$$

= {By coproduct fusion, eq. 5 *(p.42)* }

$$(\![[zero \cdot 1, plus \cdot (id \times outl)], [zero \cdot 1, succ \cdot outr \cdot (id \times outr)]]\!)$$

= {$zero$ is constant}

$$(\![[zero, plus \cdot (id \times outl)], [zero, succ \cdot outr \cdot (id \times outr)]]\!)$$

= {By definition of $\times$ (eq 4) *(p. 40, eq. 2.7)*}

$$(\![[zero, plus \cdot (id \times outl)], [zero, succ \cdot outr \cdot \langle id \cdot outl, outr \cdot outr \rangle]]\!)$$

= {Defn. of *outr* (eq 3) *(p. 41)*}

$$(\![[zero, plus \cdot (id \times outl)], [zero, succ \cdot outr \cdot outr]]\!)$$

= {Using the exchange law (eq. 6) *(p. 45)*}

$$(\!|\langle zero, zero\rangle, \langle plus \cdot (id \times outl), succ \cdot outr \cdot outr\rangle|\!)$$

Now to obtain the original definition it is sufficient to define *zeros* as it is defined above, and to formulate some pointwise function *pluss*:

$$\langle plus \cdot (id \times outl), succ \cdot outr \cdot outr\rangle\ (a, (b, n))$$

= { fusion law for *pair* (eq. 1)}

$$(plus((id \times outl)(a, (b, n))), succ(outr(outr(a, (b, n)))))$$

= { $(f \times g)\langle p, q\rangle = \langle f\ p, g\ q\rangle$ and defn of outr (eq.3)}

$$(plus(a, b), succ\ n)$$

= { defn. of *plus* and *succ* }

$$(a + b, n + 1)$$

So using this we have shown that

$$\langle sum, length\rangle = (\!|zeros, pluss|\!)$$

3 Deriving an implementation

In order to derive an implementation we will first need to understand what a catamorphism is in *listr*. To recap,

$$listrA ::= nil|cons(A, listrA)$$

therefore

$$\begin{aligned} \alpha &= [nil, cons] \\ \mathsf{F}h &= id + (id \times f) \end{aligned}$$

if h is the catamorphism $(\!|c, f|\!)$, then what function is h?

$$h = (\!|c, f|\!)$$

= { By definition of catamorphisms (*p. 46*) }

$$h \cdot \alpha = [nil, cons] \cdot \mathsf{F}h$$

= { By definition of F (eq. 8) }

$$h \cdot \alpha = [c, f] \cdot (id + (id \times h))$$

= { Fusion coproduct }

$$h \cdot \alpha = [c \cdot id, f \cdot (id \times h)]$$

= { definition of $(\times)$ }

$$h \cdot \alpha = [c \cdot id, f \cdot \langle id \cdot outl, h \cdot outr\rangle]$$

= { Definition of *id* }

$$h \cdot \alpha = [c, f \cdot \langle outl, h \cdot outr\rangle]$$

= { Definition of α }

$$h \cdot [nil, cons] = [c, f \cdot \langle outl, h \cdot outr \rangle$$

= { coproduct }

$$[h \cdot nil, h \cdot cons] = [c, f \cdot \langle outl, h \cdot outr \rangle]$$

= { Cancellation }

$$\begin{aligned} h \cdot nil &= c \\ h \cdot cons &= f \cdot \langle outl, h \cdot outr \rangle \end{aligned}$$

= { Introduce pointwise expression }

$$\begin{aligned} h\ nil &= c \\ h\ (cons\ (a, b)) &= f \cdot \langle outl, h \cdot outr \rangle (a, b) \end{aligned}$$

= { Simplify }

$$h\ nil = c \tag{10}$$

$$h\ (cons\ (a, b)) = f\ (a, h\ b) \tag{11}$$

This function is the *foldr* function familiar to functional programmers.

3.1 Sum

Using equations 10 and 11 we can derive a clear expression of the *sum* catamorphism:

$$sum = (\![zero, plus]\!)$$

= { Equations 10 and 11 }

$$\begin{aligned} sum\ nil &= zero \\ sum\ (cons(a, b)) &= plus(a, (sum\ b)) \end{aligned}$$

In a functional programming notation, *nil* is written `[]` and *cons* is an infix function written `(_:_)`. The symbol *zero* is written `0`, and *plus* is an infix function written `_+_`:

```
sum [] = 0
sum (a:b) = a+(sum b)
```

3.2 Length

$$length = (\![zero, succ \cdot outr]\!)$$

= { Equations 10 and 11 }

$$\begin{aligned} length\ nil &= zero \\ length\ (cons(a, b)) &= succ \cdot outr(a, average\ b) \end{aligned}$$

= { Definition of *outr* }

$$\begin{aligned} length\ nil &= zero \\ length\ (cons(a, b)) &= succ(average\ b) \end{aligned}$$

As before we can write this in the notation of functional programming languages. The successor function, *succ* can be written as `1+`:

```
length [] = 0
length (a:b) = 1+(length b)
```

3.3 Average

$$average = div \cdot \langle sum, length \rangle$$

= { direct substitution }

$$average\ a = div(sum\ a, length\ a)$$

Which, when written in the style of a functional program gives us:

```
average a = div (sum a,length a)
            where
              sum [] = 0
              sum (a:b) = a+(sum b)

              length [] = 0
              length (a:b) = 1+(length b)
```

Clearly this is unsatisfactory; there will be two traversals of the input list just as discussed previously. Now compare the implementation we can derive from the second version we produced:

$$average = div \cdot \langle sum, length \rangle$$

= { Banana split law }

$$average = div \cdot (\![zeros, pluss]\!)$$

= { Section 2.3 }

$$average = div \cdot (\![\langle zero, zero \rangle, \langle plus \cdot (id \times outl), succ \cdot outr \cdot outr \rangle]\!)$$

= { Setting h to be the catamorphism on the r.h.s }

$$average = div \cdot h$$

We must now derive an implementation for the catamorphism h.

$$h = (\![\langle zero, zero \rangle, \langle plus \cdot (id \times outl), succ \cdot outr \cdot outr \rangle]\!)$$

= { Equations 10 and 11 }

$$\begin{aligned} h\ nil &= \langle zero, zero \rangle \\ h\ (cons(a,b)) &= \langle plus \cdot (id \times outl), succ \cdot outr \cdot outr(a, h\ b) \rangle \end{aligned}$$

=

$$\begin{aligned} h\ nil &= \langle zero, zero \rangle \\ h\ (cons(a,b)) &= \langle plus \cdot (id \times outl)(a, h\ b), succ \cdot outr \cdot outr(a, h\ b) \rangle \end{aligned}$$

= { id, $outl$ and $outr$ }

$$\begin{aligned} h\ nil &= \langle zero, zero \rangle \\ h\ (cons(a,b)) &= \langle plus \cdot (id\ a, outl(h\ b)), succ \cdot outr(h\ b) \rangle \end{aligned}$$

This too can be written in a functional style. The functors *outl* and *outr* can be written `fst` and `snd` if we use the standard tuple notation to represent pairs (that is, parentheses surround the elements and commas separate them).

```
h [] = (0,0)
h (a:b) = (a+(fst (h b)),1+(snd (h b)))
```

We can simplify this if we introduce the notational convenience of a `where` clause. We might wish to state that within the second equation,

```
c = fst (h b)
d = snd (h b)
```

or, more succinctly

```
(c,d) = h b
```

We create such a local definition by using the word `where` on the right hand side of the equation we wish the definition to be contained in.

```
h [] = (0,0)
h (a:b) = (a+c,1+d)
          where (c,d) = h b
```

3.4 Completing the implementation

It only remains to provide an implementation for *div*.

```
div (0,0) = 0
div (a,b) = a/b
```

By substitution, the final implementation is (composition, $f \cdot g$ is written `f.g` in Haskell [1]).

```
average = div . h
          where h [] = (0,0)
                h (a:b) = (a+c,1+d)
                          where (c,d) = h b

div (a,b) = a/b
```

References

[1] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[2] Richard Bird and Oege de Moor. Untitled documents for "algebra of programming". `http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/book.html`.

[3] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1996.

[4] F. W. Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: a first introduction to categories*. Buffalo Workshop Press, Buffalo, NY, 1991.